Reminder

- In class we introduced space-complexity, defined as the maximum number of tape cells used by a Turing machine (on any computational branch for non-deterministic machines).
- \( \text{PSPACE} = \bigcup_k \text{SPACE}(n^k) \).
- \( \text{NPSPACE} = \bigcup_k \text{NSPACE}(n^k) \).
- 
  Savitch’s Theorem:
  - Statement: For any function \( f : \mathbb{N} \rightarrow \mathbb{R}^+ \) where \( f(n) \geq n \), we have \( \text{NSPACE}(f(n)) \subseteq \text{SPACE}(f^2(n)) \).
  - Corollary: \( \text{PSPACE} = \text{NPSPACE} \).
- 
  PSPACE-completeness: A language \( B \) is PSPACE-complete if:
  1. \( B \in \text{PSPACE} \)
  2. \( A \leq_P B \) \( \forall A \in \text{PSPACE} \)

Example 1 — PSPACE and NPSPACE algorithms for \( LADDER_{DFA} \)

A word ladder is a sequence of strings \( s_1, s_2, ..., s_n \) where every string differs from the previous string by exactly one symbol. For example, the following sequence is a word ladder starting with “lead” and ending with “gold”:

\[
\text{lead} \rightarrow \text{mead} \rightarrow \text{meld} \rightarrow \text{mold} \rightarrow \text{gold}
\]

Define the language \( LADDER_{DFA} = \{ \langle M, s, t \rangle \mid M \text{ is a DFA and } L(M) \text{ contains a word ladder beginning with } s \text{ and ending with } t \} \).

1. Give an NPSPACE algorithm that decides \( LADDER_{DFA} \).
2. Give an PSPACE algorithm that decides \( LADDER_{DFA} \).

Solutions

1. Given \( \langle M, s, t \rangle \) ...

   - Starting at \( s \), nondeterministically change one symbol at a time and check if the new word is in \( L(M) \).
   - If it isn’t, reject, and otherwise continue.
• If we ever reach $t$, accept.
• If we’ve considered more than $|\Sigma|^n$ steps (where $n = |s|$), then reject.

Each thread of this algorithm clearly uses polynomial space. To make this a decider, we just have to make sure no thread of this algorithm runs forever. We know that if any accepting thread exists, then there must be some accepting thread that doesn’t visit the same word twice. Since the number of words in $L(M)$ of length $n$ is at most $|\Sigma|^n$, we can terminate a thread when it runs for longer than this many steps, since such a thread must have visited a word twice.

2. Notice that if a word ladder exists from $s$ to $t$, then there is a word ladder from $s$ to $t$ that uses a maximum of $|\Sigma|^n$ words. That means that there is some word $r$ where we can get from $s$ to $r$ and from $r$ to $t$ both in at most $\frac{k}{2}|\Sigma|^n$ steps. We can derive a recursive algorithm from this as follows:

Consider the general problem of deciding whether we can get from some word $u$ to $v$ in at most $k$ steps. To solve this . . .

• If $k = 1$, check that $u \to v$ is a valid transition.
• If $k > 1$, then for each word $w \in \Sigma^n$:
  – Check if we can get from $u$ to $w$ and from $w$ to $v$ both in at most $\frac{k}{2}$ steps.
  – If both checks succeed, then accept.

Now, given $(M, s, t)$ . . .

• Run the above recursive algorithm with $u = s$, $v = t$, and $k = |\Sigma|^n$
• Accept if the recursive algorithm accepts, otherwise reject.

From the reasoning above we can determine that our algorithm decides $LADDER_{DFA}$. Now all we need is to show that our algorithm runs in polynomial space. Notice that our maximum recursion depth is $\log_2(|\Sigma|^n) = n \log_2(|\Sigma|) = O(n)$. In each recursive layer we need to keep track of a single string $w$ of length $n$, and the computations in each layer require $O(n)$ space. This means that each recursive layer uses $O(n)$ space, so the total space needed is $O(n^2)$.

**Example 2 — $TBQF$ is PSPACE-complete**

Recall that $TQBF = \{ \langle \phi \rangle \mid \phi$ is a true fully-quantified boolean formula$\}$. Show that $TQBF$ is PSPACE-complete.

**Solution**

First, we need to show that $TQBF \in \text{PSPACE}$.

Given input $\langle \phi \rangle$ . . .

• If $\phi$ contains no quantifiers, then it is an expression with only constants, so evaluate $\phi$ and accept if it is true; otherwise, reject.
• If $\phi = \exists x \psi$ for some formula $\psi$, recursively call T on $\psi$, first with 0 substituted for $x$ and then with 1 substituted for $x$. If either result is accept, then accept; otherwise, reject.

• If $\phi = \forall x \psi$ for some formula $\psi$, recursively call T on $\psi$, first with 0 substituted for $x$ and then with 1 substituted for $x$. If both results are accept, then accept; otherwise, reject.

We need to show that our algorithm runs in polynomial space. Note that the depth of the recursion is at most the number of variables. At each level we need only store the value of one variable, so the total space used is $O(n)$, where $n$ is the number of variables that appear in $\phi$.

Now let $A$ be a language decidable by some Turing machine $M$ in space $n^k$ for some constant $k$. To show that $TQBF \in \text{PSPACE}$, we will show that $A \leq_p TQBF$. Given some $w$, we want to construct $\phi$ such that $w \in A \iff \phi \in TQBF$.

Let each configuration of $M$ have a variable for each tape symbol and state, corresponding to their possible settings. Then each configuration has $n^k$ cells and so is encoded by $O(n^k)$ variables. Given two configurations $c_a, c_b$ representing two collections of variables and some integer $t > 0$, we want to create a formula $\phi_{a,b,t}$ such that $\phi_{a,b,t}$ is true iff $M$ can go from $c_a$ to $c_b$ in at most $t$ steps. The idea is to then have $\phi = \phi_{\text{start}, \text{accept}, h}$, where $h = 2^{(dn^k)}$ with $d$ chosen such that $M$ has no more than $2^{(dn^k)}$ possible configurations on an input of length $n$.

For $t = 1$, we design $\phi_{a,b,1}$ to evaluate to true only if $c_a = c_b$ or $c_b$ follows from $c_a$ in a single step of $M$. For the first possibility, we express the equality by writing a boolean expression saying that each of the variables representing $c_a$ contains the same Boolean value as the corresponding variable representing $c_b$. For the second possibility, we can express that $c_a$ yields $c_b$ in a single step of $M$ by writing boolean expressions stating that the contents of each triple of $c_a$’s cells correctly yields the contents of the corresponding triple of $c_b$’s cells.

For $t > 1$, we recursively construct $\phi_{a,b,1}$. Notice that if we can go from $c_a$ to $c_b$ in at most $t$ steps, then there exists some configuration $c_d$ such that we can go from $c_a$ to $c_d$ and $c_d$ to $c_a$ in at most $\frac{t}{2}$ steps each. One possible way to write this formula is $\phi_{a,b,t} = \exists d(\phi_{a,d,\frac{t}{2}} \land \phi_{d,b,\frac{t}{2}})$. However, the size of the formula doubles at each stage of the recursion, which for $t = h$ gives us a formula of size $t = h = 2^{(dn^k)}$, which is exponential with respect to $n$. Alternatively, we can write the equivalent formula $\phi_{a,b,t} = \exists d \forall u \forall v((u = a \land v = d) \lor (u = d \land v = b))[\phi_{a,u,\frac{t}{2}}]$. Finally, we need to show our reduction is in polynomial time. Each stage of the recursion runs in $O(n^k)$, and the maximum recursion depth is $\log_2 h = O(n^k)$. Thus our reduction runs in $O(n^2k)$ time, and therefore we have $A \leq_p TQBF$. We conclude that $TQBF$ is PSPACE-complete.

Discussion — Why do we do poly-time (P) reductions?

When showing a language is NP-complete or PSPACE-complete we’ve used polynomial time reductions. Why don’t we use NP reductions or PSPACE reductions?
Let’s go over the basics of reductions again. In general, a reduction from some language $A$ to some language $B$ is a function $f : \Sigma^* \rightarrow \Sigma^*$ such that for all $w \in \Sigma^*$, $w \in A \iff f(w) \in B$. If we have $A \leq B$, this tells roughly that solving $B$ is at least as hard as solving $A$.

For mapping-reducibility we added the condition that $f$ be Turing-computable. Notice that for mapping-reducibility, all (except for $\Sigma^*$ and $\emptyset$) decidable problems are reducible to each other. Essentially, given that $B \neq \Sigma^*, \emptyset$ we know that there exists $x, y \in \Sigma^*$ such that $x \in B$ and $y \notin B$. Then our reduction is simply

$$f(w) = \begin{cases} x & w \in A \\ y & w \notin A \end{cases}$$

Computing $f(w)$ simply involves using the decider for $A$, so the reduction is Turing-computable.

For polynomial-reducibility we added the condition that $f$ be computable in polynomial time. Similarly for polynomial-reducibility, all (except for $\Sigma^*$ and $\emptyset$) problems in P are reducible to each other.

In fact, given some class of languages $C$, all non-trivial languages in $C$ are $C$-reducible to each other; the reduction simply solves the problem in question and produces an instance with the same value. This means that “completeness” is only meaningful if you use a class of reductions from a potentially smaller (or weaker) class. Take PSPACE for example; it wouldn’t make any sense to do PSPACE reductions. You could potentially use other reductions, however in practice our reductions are simple transformations that are easily doable in polynomial-time.

By convention, we use polynomial-time reductions for classes within the polynomial hierarchy. However, when studying classes contained within P we use log-space reductions, as we will see later on in the course.